Listas

Sergio Canuto
sergio.canuto@ifg.edu.br

Relembrando Tipo Abstrato de Dados (TAD)...

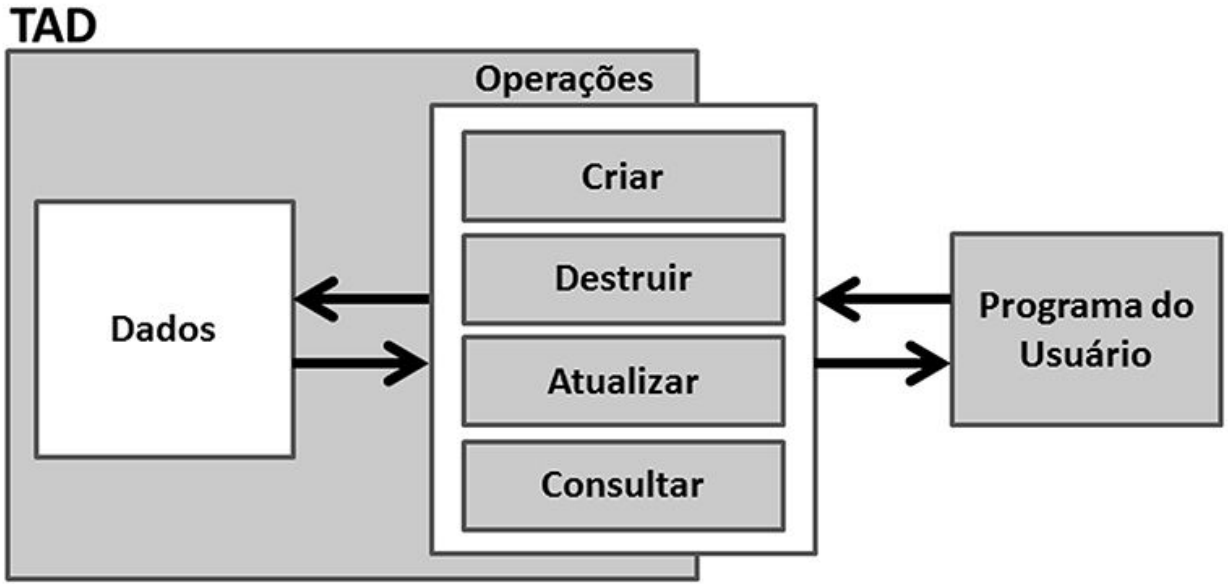
Conceituações - Tipo Abstrato de dados

- Um **tipo de dado** define o conjunto de **valores** (domínio) e **operações** que uma variável pode assumir.
  - Ex.: Int, Char, float, etc…
- Uma **estrutura de dados** consiste em um conjunto de tipos de dados em que existe algum tipo de relacionamento lógico estrutural.
  - Ex.: Na linguagem C temos os **array**, **struct**, **union** e **enum**, todas criadas a partir dos tipos de dados básicos.
- Um **tipo abstrato de dados**, ou **TAD**, é um conjunto de dados estruturados e as operações que podem ser executadas sobre esses dados.
- Tanto a representação quanto as operações do **TAD** são especificadas pelo programador.
  - O usuário utiliza o **TAD** como uma caixa-preta por meio de sua **interface**.

Tipo Abstrato de dados - Operações Básicas

- Tipos abstratos de dados incluem as operações para a manipulação de seus dados. Essas operações variam de acordo com o **TAD** criado, porém as seguintes operações básicas são possíveis:
  - Criação do **TAD**.
  - Inserção de um novo elemento no **TAD**.
  - Remoção de um elemento do **TAD**.
  - Acesso a um elemento do **TAD**.
  - Destruição do **TAD**.

Tipo Abstrato de dados



Tipo Abstrato de dados

O uso de um TAD traz uma série de vantagens:

- Encapsulamento: ao ocultarmos a implementação, fornecemos um conjunto de operações possíveis para o TAD.
- Segurança: o usuário não tem acesso direto aos dados. Isso evita que ele manipule os dados de uma maneira imprópria.
- Flexibilidade: podemos alterar o TAD sem alterar as aplicações que o utilizam.
- Reutilização: a implementação do TAD é feita em um módulo diferente do programa do usuário.

Modularização & TAD

- Na linguagem C, é comum definirmos ao menos 3 arquivos:
  - Um para interface (arquivoexemplo.h)
  - Outro para implementação da interface (arquivoexemplo.c)
  - Outro para a função main (main.c)

Exemplo de TAD - Ponto

Arquivo Ponto.h

```
01  typedef struct ponto Ponto;
02  //Cria um novo ponto
03  Ponto* Ponto_cria(float x, float y);
04  //Libera um ponto
05  void Ponto_libera(Ponto* p);
06  //Acessa os valores "x" e "y" de um ponto
07  int Ponto_acessa(Ponto* p, float* x, float* y);
08  //Atribui os valores "x" e "y" a um ponto
09  int Ponto_atribui(Ponto* p, float x, float y);
10  //Calcula a distância entre dois pontos
11  float Ponto_distancia(Ponto* p1, Ponto* p2);
```

Arquivo Ponto.c

```
01  #include <stdlib.h>
02  #include <math.h>
03  #include "Ponto.h" //inclui os Protótipos
04  struct ponto{//Definição do tipo de dados
05    float x;
06    float y;
07  };
```

Exemplo de TAD - Ponto

Criando um ponto

```
01  Ponto* Ponto_cria(float x, float y){
02      Ponto* p = (Ponto*) malloc(sizeof(Ponto));
03      if(p != NULL){
04          p->x = x;
05          p->y = y;
06      }
07      return p;
08  }
```

Destruindo um ponto

```
01  void Ponto_libera(Ponto* p){
02      free(p);
03  }
```

Exemplo de TAD - Ponto

Atribuindo um valor ao ponto

```
01  int Ponto_atribui(Ponto* p, float x, float y){
02      if(p == NULL)
03          return 0;
04      p->x = x;
05      p->y = y;
06      return 1;
07  }
```

Calculando a distância entre dois pontos

```
01  float Ponto_distancia(Ponto* p1, Ponto* p2){
02      if(p1 == NULL || p2 == NULL)
03          return -1;
04      float dx = p1->x - p2->x;
05      float dy = p1->y - p2->y;
06      return sqrt(dx * dx + dy * dy);
07  }
```

Exemplo de TAD - Ponto

Exemplo: utilizando o TAD ponto

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "Ponto.h"
04  int main(){
05      float d;
06      Ponto *p,*q;
07      //Ponto r; //ERRO
08      p = Ponto_cria(10,21);
09      q = Ponto_cria(7,25);
10      //q->x = 2; //ERRO
11      d = Ponto_distancia(p,q);
12      printf("Distancia entre pontos: %f\n",d);
13      Ponto_libera(q);
14      Ponto_libera(p);
15      system("pause");
16      return 0;
17  }
```

Listas com TAD

Listas - Definição

- O conceito de lista é algo muito comum para as pessoas. Trata-se de um relação finita de itens, todos eles contidos dentro de um mesmo tema.
- Vários são os exemplos possíveis de listas: itens em estoque em uma empresa, dias da semana, lista de compras do supermercado, convidados de uma festa etc.
- Em ciência da computação, uma lista é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador. Uma estrutura do tipo lista é uma sequência de elementos do mesmo tipo.
- Seus elementos possuem estrutura interna abstraída, ou seja, sua complexidade é arbitrária e não afeta o seu funcionamento.

Listas

33	23	16	15	43	58

Tipos de Listas

Quanto à inserção/remoção de elementos da lista, temos:

- **Lista convencional**: pode ter elementos inseridos ou removidos de qualquer lugar dela.
- **Fila**: estrutura do tipo FIFO (First In First Out), os elementos só podem ser inseridos no final, e acessados ou removidos do início da lista.
- **Pilha**: estrutura do tipo LIFO (Last In First Out), os elementos só podem ser inseridos, acessados ou removidos do final da lista.

Tipos de Listas - alocação

Quanto à alocação de memória, podemos utilizar **alocação estática** ou **dinâmica** para implementar uma lista:

- **Alocação estática**: o espaço de memória é alocado no momento da compilação do programa. É necessário definir o número máximo de elementos que a lista irá possuir.
- **Alocação dinâmica**: o espaço de memória é alocado em tempo de execução. A lista cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos.

Tipos de Listas - Acesso

E, independentemente de como a memória foi alocada, podemos acessar os seus elementos de duas formas:

- **Acesso sequencial**: os elementos são armazenados de forma consecutiva na memória (como em um array ou vetor). A posição de um elemento pode ser facilmente obtida a partir do início da lista.
- **Acesso encadeado**: cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas. É necessário que cada elemento da lista armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na lista.

OPERAÇÕES BÁSICAS DE UMA LISTA

Independentemente do tipo de alocação e acesso usado na implementação de uma lista, as seguintes operações básicas (TAD) são possíveis:

- Criação da lista.
- Inserção de um elemento na lista.
- Remoção de um elemento da lista.
- Busca por um elemento da lista.
- Destruição da lista.
- Além de informações com tamanho, se a lista está cheia ou vazia.

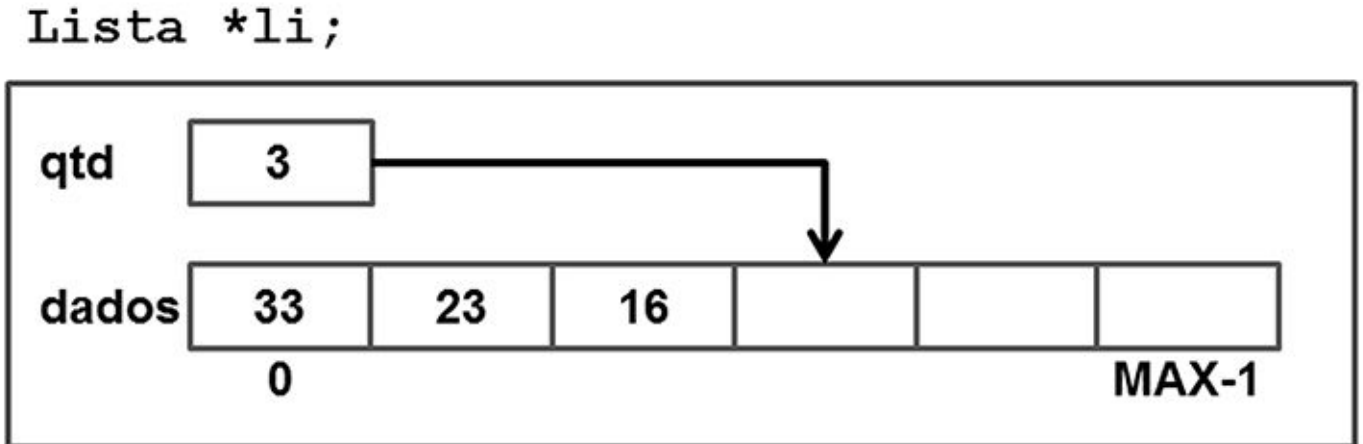
Inserção/Remoção em uma LISTA

- Três tipos de inserção/remoção:
  - No início, no final ou no meio (isto é, entre dois elementos) da lista.
- A inserção no meio da lista é comumente usada quando se deseja inserir um elemento de forma ordenada na lista.
- A remoção do meio da lista é comumente usada quando se deseja remover um elemento específico da lista.

LISTA SEQUENCIAL ESTÁTICA

LISTA SEQUENCIAL ESTÁTICA

- Uma **lista sequencial estática** ou **lista linear estática** é uma lista definida utilizando alocação estática e acesso sequencial dos elementos. Trata-se do tipo mais simples de lista possível. Ela é definida utilizando um array, de modo que o sucessor de um elemento ocupa a posição física seguinte deste.
- Ex.: Acesso apenas a um ponteiro do tipo lista:



lista sequencial estática tem vantagens e desvantagens

Vantagens em se definir uma lista utilizando um array:

- Acesso rápido e direto aos elementos (índice do array).
- Tempo constante para acessar um elemento.
- Facilidade para modificar as suas informações.

Desvantagens de listas com uso de arrays:

- Definição prévia do tamanho do array e, consequentemente, da lista.
- Dificuldade para inserir e remover um elemento entre outros dois: é necessário deslocar os elementos para abrir espaço dentro do array.

lista sequencial estática tem vantagens e desvantagens

Em geral, usamos esse tipo de lista nas seguintes situações:

- Listas pequenas.
- Inserção e remoção apenas no final da lista.
- Tamanho máximo da lista bem definido.
- A busca é a operação mais frequente.

Definindo o tipo lista sequencial estática

Vamos começar definindo o arquivo **ListaSequencial.h**. Por se tratar de uma lista estática, temos que estabelecer:

- O tamanho máximo do array utilizado na lista, representada pela constante **MAX** (linha 1).
- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 2-6).
- Para fins de padronização, um novo nome para o tipo lista (linha 7). Esse é o tipo que será usado sempre que se desejar trabalhar com a lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 9-21) e que serão implementadas no arquivo **ListaSequencial.c**

Definindo o tipo lista sequencial estática

Arquivo ListaSequencial.h

```
01  #define MAX 100
02  struct aluno{
03      int matricula;
04      char nome[30];
05      float n1,n2,n3;
06  };
07  typedef struct lista Lista;
08
09  Lista* cria_lista();
10  void libera_lista(Lista* li);
11  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
12  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
13  int insere_lista_final(Lista* li, struct aluno al);
14  int insere_lista_inicio(Lista* li, struct aluno al);
15  int insere_lista_ordenada(Lista* li, struct aluno al);
16  int remove_lista(Lista* li, int mat);
17  int remove_lista_inicio(Lista* li);
18  int remove_lista_final(Lista* li);
19  int tamanho_lista(Lista* li);
20  int lista_cheia(Lista* li);
21  int lista_vazia(Lista* li);
```

Definindo o tipo lista sequencial estática (cont)

As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).

•A definição do tipo que descreve o funcionamento da lista, **struct lista** (linhas 5-8).

•As implementações das funções definidas no arquivo **ListaSequencial.h**. As implementações dessas funções serão vistas a seguir...

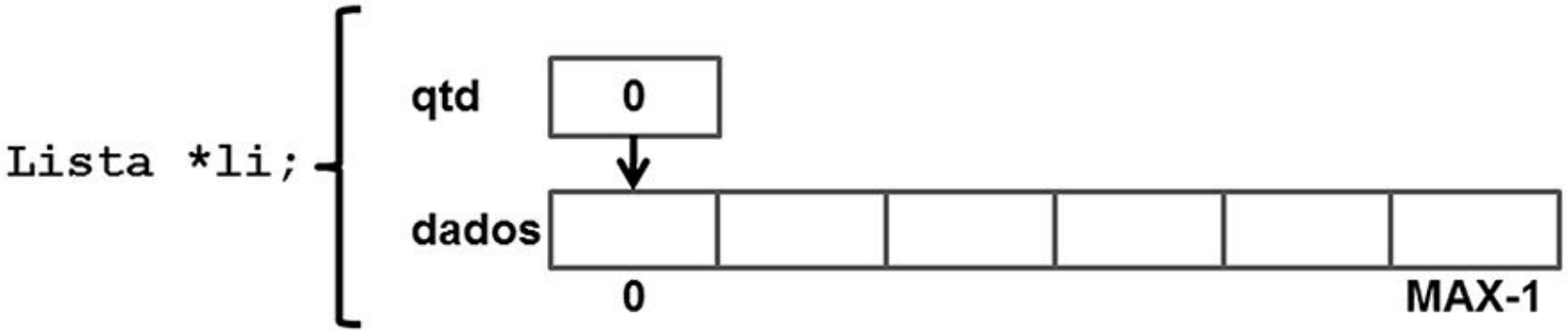
Arquivo ListaSequencial.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaSequencial.h" //inclui os protótipos
04  //Definição do tipo lista
05  struct lista{
06      int qtd;
07      struct aluno dados[MAX];
08  };
```

Criando e destruindo uma lista

Criando uma lista

```
01  Lista* cria_lista(){
02      Lista *li;
03      li = (Lista*) malloc(sizeof(struct lista));
04      if(li != NULL)
05          li->qtd = 0;
06      return li;
07  }
```

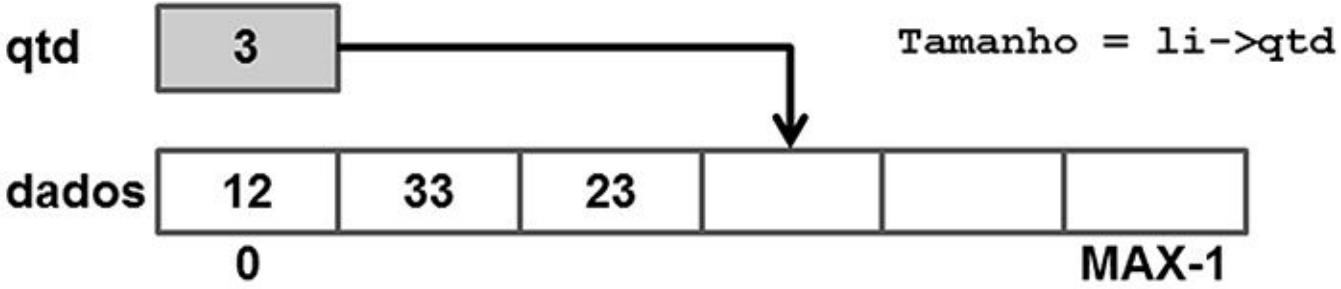


Criando e destruindo uma lista

Destruindo uma lista

```
01  void libera_lista(Lista* li){
02      free(li);
03  }
```

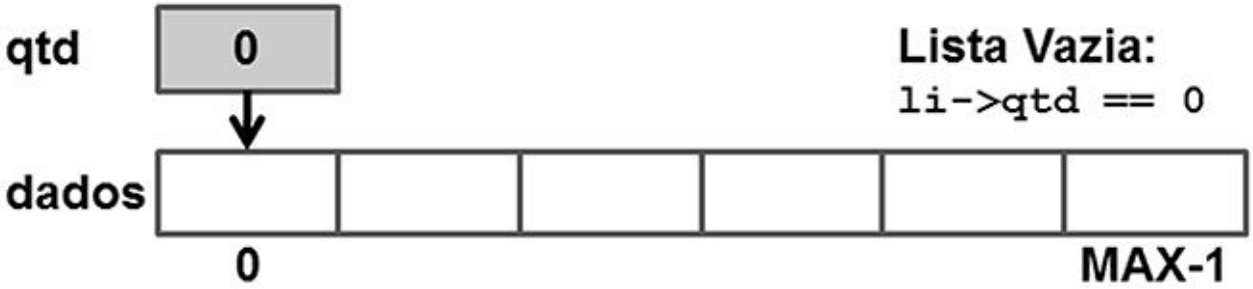
Informações básicas sobre a lista - tamanho



Tamanho da lista

```
01  int tamanho_lista(Lista* li){
02      if(li == NULL)
03          return -1;
04      else
05          return li->qtd;
06  }
```

Informações básicas sobre a lista - lista vazia



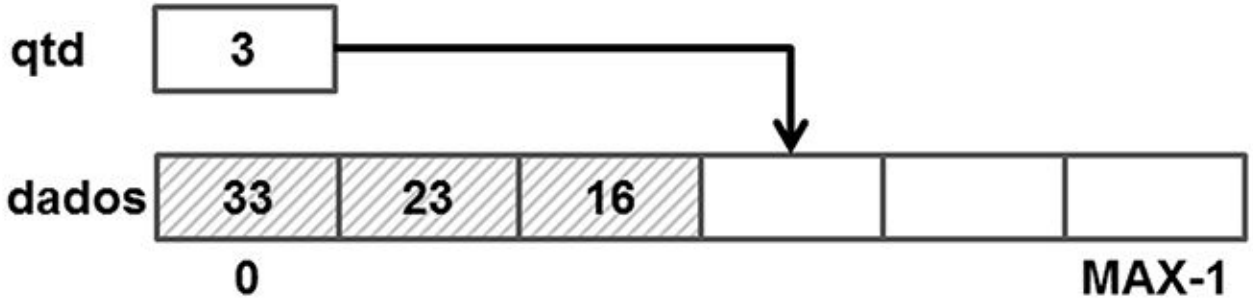
Retornando se a lista está vazia

```
01  int lista_vazia(Lista* li){
02      if(li == NULL)
03          return -1;
04      return (li->qtd == 0);
05  }
```

Inserindo um elemento na lista - no início

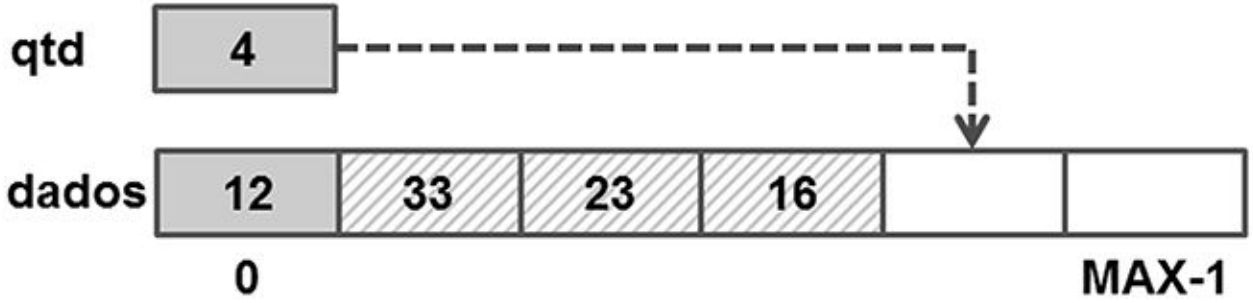
Inserindo um elemento no início da lista

```
01  int insere_lista_inicio(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      if(li->qtd == MAX)//lista cheia
05          return 0;
06      int i;
07      for(i=li->qtd-1; i>=0; i--)
08          li->dados[i+1] = li->dados[i];
09      li->dados[0] = al;
10      li->qtd++;
11      return 1;
12  }
```



Desloca os elementos uma posição para frente e insere:

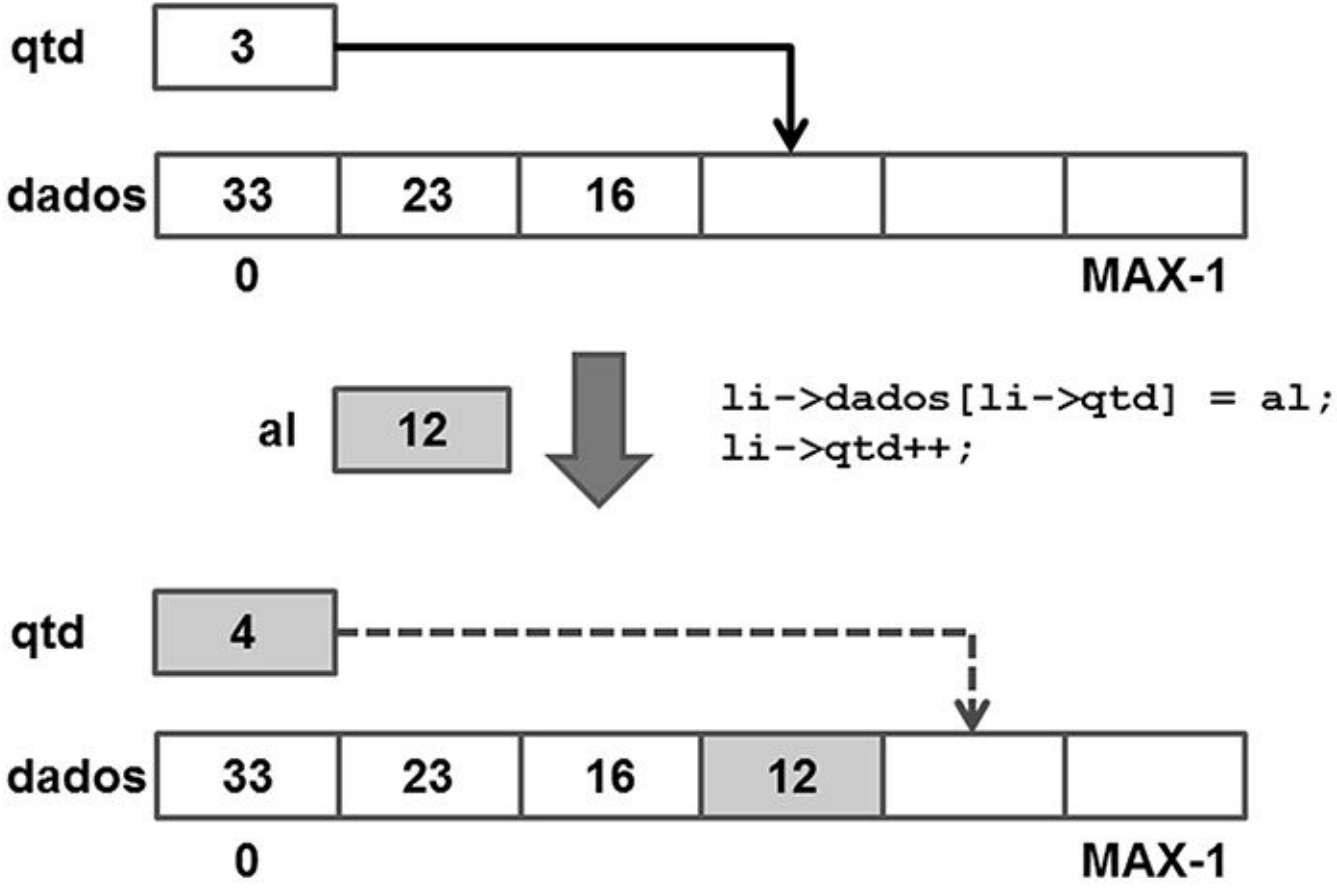

```
for(i=li->qtd-1; i>=0; i--)
   li->dados[i+1] = li->dados[i];
li->dados[0] = al;
li->qtd++;
```



Inserindo um elemento na lista - no final

Inserindo um elemento no final da lista

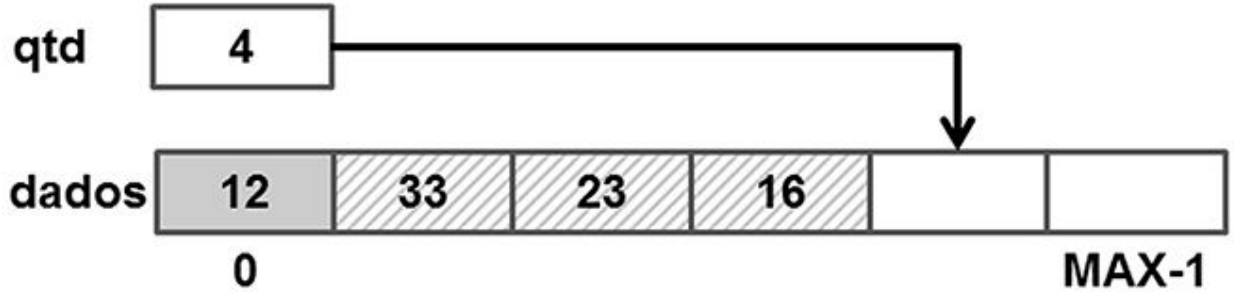
```
01  int insere_lista_final(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      if(li->qtd == MAX)//lista cheia
05          return 0;
06      li->dados[li->qtd] = al;
07      li->qtd++;
08      return 1;
09  }
```


qtd
3
dados
33
23
16
0
MAX-1
al
12
li->dados[li->qtd] = al;
li->qtd++;
qtd
4
dados
33
23
16
12
0
MAX-1

Removendo do início da lista

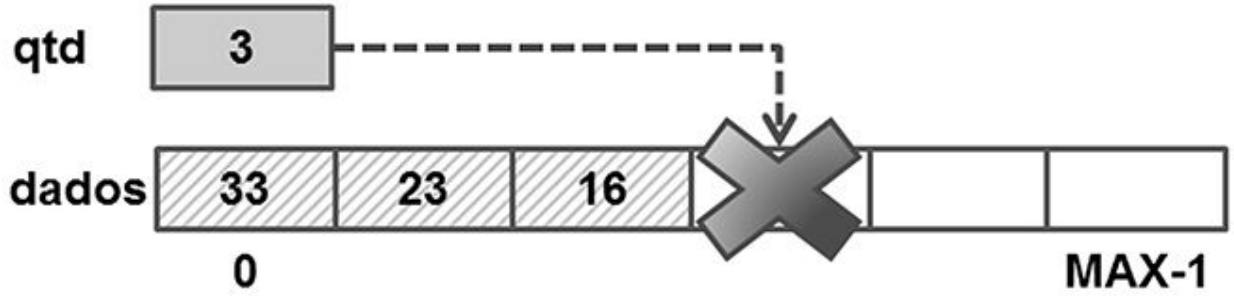
Removendo um elemento do início da lista

```
01  int remove_lista_inicio(Lista* li){
02      if(li == NULL)
03          return 0;
04      if(li->qtd == 0)//lista vazia
05          return 0;
06      int k = 0;
07      for(k=0; k< li->qtd-1; k++)
08          li->dados[k] = li->dados[k+1];
09      li->qtd--;
10      return 1;
11  }
```



Desloca os elementos uma posição para trás:

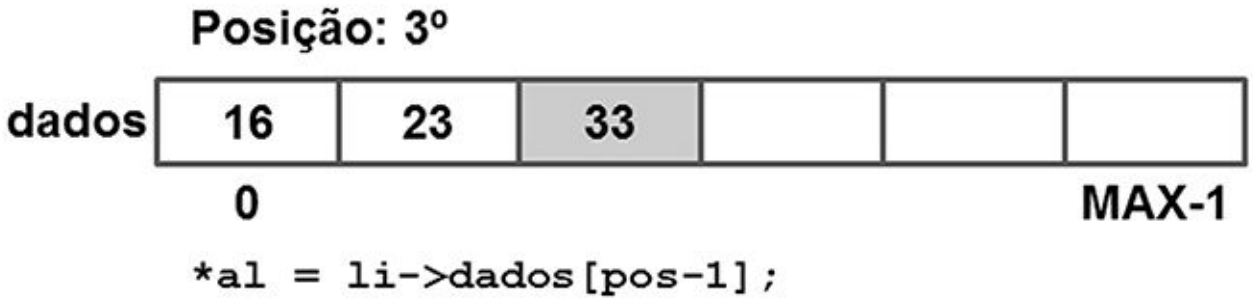
```
for(k=0; k< li->qtd-1; k++)
   li->dados[k] = li->dados[k+1];
li->qtd--;
```



busca por elemento

Busca um elemento por posição

```
01  int busca_lista_pos(Lista* li,int pos,struct aluno *al){
02      if(li == NULL || pos <= 0 ||  pos > li->qtd)
03          return 0;
04      *al = li->dados[pos-1];
05      return 1;
06  }
```

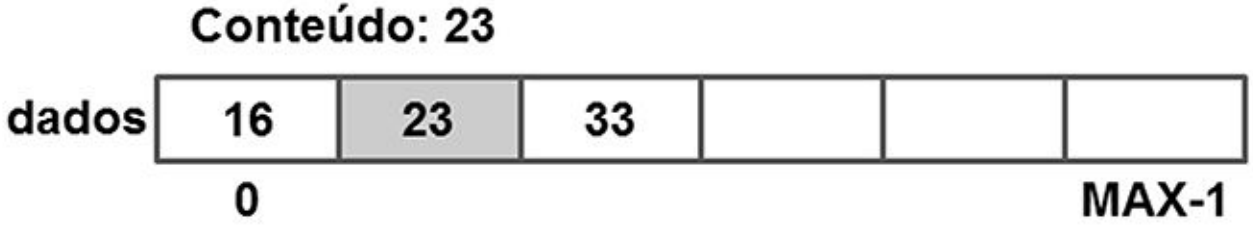


busca por elemento

Busca um elemento por conteúdo

```
01  int busca_lista_mat(Lista* li, int mat, struct aluno *al){
02      if(li == NULL)
03          return 0;
04      int i = 0;
05      while(i<li->qtd && li->dados[i].matricula != mat)
06          i++;
07      if(i == li->qtd)//elemento não encontrado
08          return 0;
09
10      *al = li->dados[i];
11      return 1;
12  }
```

busca por elemento - conteúdo



Busca pelo elemento:

```
while(i<li->qtd && li->dados[i].matricula != mat)
        i++;
```

Achou o elemento:

```
*al = li->dados[i];
```

LISTA DINÂMICA ENCADEADA

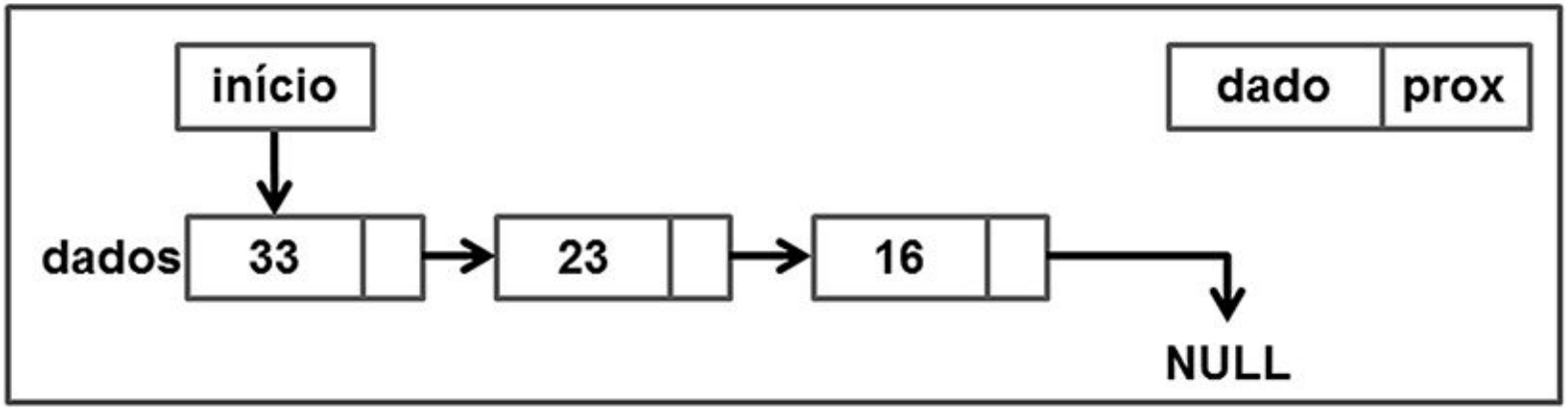
LISTA DINÂMICA ENCADEADA

- Uma **lista dinâmica encadeada** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e tem sua memória liberada, à medida que é removido.
- Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação:
  - um campo de **dado**, utilizado para armazenar a informação inserida na lista, e
  - um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na lista.

LISTA DINÂMICA ENCADEADA

- Uma **lista dinâmica encadeada** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e tem sua memória liberada, à medida que é removido.
- Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação:
  - um campo de **dado**, utilizado para armazenar a informação inserida na lista, e
  - um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na lista.

Lista *li



Vantagens/desvantagens: lista dinâmica encadeada

Várias são as vantagens de se definir uma lista utilizando uma abordagem dinâmica e encadeada:

- Melhor utilização dos recursos de memória.
- Não é preciso definir previamente o tamanho da lista.
- Não se precisa movimentar os elementos nas operações de inserção e remoção.

Infelizmente, esse tipo de implementação também tem suas desvantagens:

- Acesso indireto aos elementos.
- Necessidade de percorrer a lista para acessar determinado elemento.

Considerando suas vantagens e desvantagens, quando devo utilizar uma **lista dinâmica encadeada**?

Em geral, usamos esse tipo de lista nas seguintes situações:

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.

lista dinâmica encadeada: ponteiros

- **lista sequencial estática** precisamos declarar apenas um ponteiro para manipular a lista
- **lista dinâmica encadeada** todos os seus elementos são ponteiros alocados dinamicamente e de forma independente

Essa pequena diferença de implementação faz com que a passagem de uma **lista dinâmica encadeada** para uma função tenha que ser feita utilizando um **ponteiro para ponteiro** em vez de um simples **ponteiro**.

Lista Encadeada

Arquivo ListaDinEncad.h

```
01  struct aluno{
02      int matricula;
03      char nome[30];
04      float n1,n2,n3;
05  };
06  typedef struct elemento* Lista;
07
08  Lista* cria_lista();
09  void libera_lista(Lista* li);
10  int insere_lista_final(Lista* li, struct aluno al);
11  int insere_lista_inicio(Lista* li, struct aluno al);
12  int insere_lista_ordenada(Lista* li, struct aluno al);
13  int remove_lista(Lista* li, int mat);
14  int remove_lista_inicio(Lista* li);
15  int remove_lista_final(Lista* li);
16  int tamanho_lista(Lista* li);
17  int lista_vazia(Lista* li);
18  int lista_cheia(Lista* li);
19  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
20  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
```

- **não precisa definir a quantidade MAX de elementos**
- **a Lista não é mais um array de dados, mas um ponteiro para um elemento.**

Lista Encadeada

Arquivo ListaDinEncad.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaDinEncad.h" //inclui os protótipos
04  //Definição do tipo lista
05  struct elemento{
06      struct aluno dados;
07      struct elemento *prox;
08  };
09  typedef struct elemento Elem;
```

Atividade (continuação da última aula)

Vetor de pontos

Utilizando os conceitos de ponteiros e ponteiros de ponteiros, modifique o código (em http://scanuto.com/vetorpontos.zip) implementando as funções cria_vetor_pontos, imprime_media e libera_vetor_pontos para tornar o código abaixo funcional:

```c
#include <stdio.h>
#include <stdlib.h>
#include "Ponto.h"
int main(){
    printf("criando vetor de 4 posicoes\n");

    cria_vetor_pontos(4);

    printf("inserindo 4 pontos\n");
    pto_insere(1,2);
    pto_insere(2,4);
    pto_insere(3,6);
    pto_insere(4,8);


    //imprime o ponto médio
    imprime_media();

    libera_vetor_pontos();

    return 0;
}
```

Vetor de pontos

Utilizando os conceitos de ponteiros e ponteiros de ponteiros, modifique o código (em http://scanuto.com/vetorpontos.zip) implementando as funções cria_vetor_pontos, imprime_media e libera_vetor_pontos para tornar o código abaixo funcional:

```
1 #include <stdlib.h>
2 #include <math.h>
3 #include "Ponto.h" //inclui os Protótipos
4 #include <stdlib.h>
5 #include <stdio.h>
6
7 //Definição do tipo de dados
8 struct ponto{
9   float x;
10  float y;
11 };
12
13 //Cria um vetor de pontos do tipo *Ponto
14 Ponto **vecpontos; //vetor de armazenamento dos ponteiros dos pontos
15 int pontos_inseridos; //quantidade pontos inseridos ate o momento
16
17 //inicializa o vetor de pontos, alocando a memoria
18 void cria_vetor_pontos(int tamanho){
19 //preencher aqui com seu codigo...
20 }
21
22 void pto_insere(float x, float y){
23 //preencher aqui com seu codigo...
24 }
25
26 //Libera toda a memoria alocada
27 void libera_vetor_pontos(){
28 //preencher aqui com seu codigo...
29 }
30
31 //imprime o ponto medio da lista de pontos
32 void imprime_media(){
33 //insira aqui o seu codigo...
34 }
```

lista de pontos

- O que colocar no lugar de “preencher aqui?”
- O código anterior segue os princípios da TAD? Se não, o que precisaríamos fazer?
  - Como modificar o código para implementar um TAD lista?
- Como “crescer” a lista para um tamanho maior que o escolhido inicialmente?
- Como implementar a função abaixo?
  - void pto_remove(float posicao_ponto)

Referências

Estrutura de Dados descomplicada em Linguagem C (André Backes): Cap 7;

Vídeo aulas:
https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/

Implementações:

http://www.facom.ufu.br/~backes/wordpress/ListaSequencial.zip

https://programacaodescomplicada.wordpress.com/complementar/